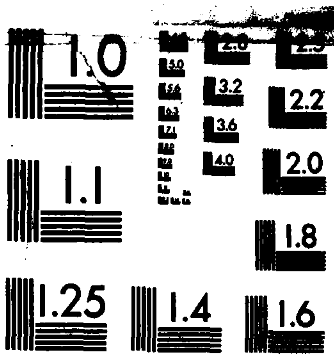
AD-A157 311 DIRECTLY EXECUTED LANGUAGES(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB M J FLYNN 10 MAY 85 ARO-18553.1-EL DAAG29-82-K-0109 1/1

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

AD-A157 311

DTIC FILE COPY

2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
REPORT NUMBER: ARO 18553.1-EL	2. GOVT ACCESSION NO.: N/A	3. RECIPIENT'S CATALOG NUMBER: N/A
TITLE (and Subtitle): "Directly Executed Languages"		5. TYPE OF REPORT & PERIOD COVERED: Final Report 19 Apr 82 - 30 Jun 85
		6. PERFORMING ORG. REPORT NUMBER
AUTHOR(s): Prof. M. Flynn		8. CONTRACT OR GRANT NUMBER(s): DAAG29-82-K-0109
PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Systems Lab, Stanford University, Stanford, CA 94305-2192		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709		12. REPORT DATE: 10 May 85
		13. NUMBER OF PAGES: 12
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE AUG 5 1985 S D A

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer architecture, instruction sets, instruction bandwidth, concurrent execution, directly executed language, direct correspondence architecture.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Computer architectures can be designed to be in close correspondence with high level computer languages. Techniques for designing this correspondence have been developed which produce instruction sets called Direct Correspondence Architectures. DCA representations mimimize the number of bits needed to encode an instruction, as well as many of the dynamic parameters associated with program execution.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

85 7 25 059

ARO 18553.1-EL

DIRECTLY EXECUTED LANGUAGES

Final Report

Prepared for

Department of the Army
U.S. Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27709

Contract No. DAAG29-82-K-0109

April 19, 1982—April 18, 1985

by

Michael J. Flynn
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, CA 94305-2192

Table of Contents

Abstract

Computer architectures or instruction sets can be designed to be in close correspondence with high level computer languages. Techniques for designing this correspondence have been developed which produce instruction sets called Direct Correspondence Architectures, or DCAs. DCA representations minimize the number of bits needed to encode an instruction, as well as minimizing many of the dynamic parameters associated with program execution. In a Pascal-based DCA, the following reductions were achieved for a broad range of benchmarks when compared to a breadbasket of conventional architectures such as S/370, VAX, and P-code:

1) P instruction bandwidth reduction: 3.46

2) I data read reduction (in bytes): 5.42

3) I data write reduction (in bytes): 14.72

A microprocessor based implementation of a Pascal-based DCA has begun. Issues in concurrency detection for these and other architectures have been investigated.

(**Keywords** include :)

Computer architecture, instruction sets, instruction bandwidth, conncurrent execution, direct correspondence architecture, directly executed languages.

1 Direct Correspondence Architecture

A natural way to make compilation as straightforward as possible is to make the instruction set or architecture fit the high-level language. This work centers on a family of architectures called direct correspondence architectures, or DCAs. High-level language statements are closely represented by DCA instructions. DCA representations minimize the number of bits needed in the instruction stream for operand specification, without resorting to encodings that require knowledge of the frequency of occurrence of individual operands. A Pascal-to-DCA compiler and a DCA processor emulator have been used to measure the number of instructions required to run test programs to completion. These results show that it is possible to make an architecture suitable for a high-level language in a way that results in architectural measures that indicate a higher speed of execution and a lower cost of implementation than some familiar architectures.

1.1 An Evaluation of Adept—A Pascal Based Architecture

This study developed and extended techniques to provide architectural correspondence between high level language objects and hardware resources so as to minimize the execution time parameters (memory traffic, program size, etc.). A resulting Pascal-based architecture (or instruction set) called Adept has been emulated, and a compiler developed for it. This allows a statistical basis for understanding the dynamic behavior of Pascal-type programs. While the study is restricted to Pascal, the resulting data is generally applicable to many familiar high level language execution environments. Data indicate that significant bandwidth reductions are possible compared to S/370, VAX, P-code, etc; specifically:

- instruction bandwidth reduction: 3.46
- data read reduction (in bytes): 5.42
- data write reduction (in bytes): 14.72

While Adept uses a bit-variable encoding of objects, most of the above reductions are retained using block encoding of, e.g., 6/12/. . . bits per object. The tradeoff between format sets and object encoding was developed.

The contour model for variable accessing (used in Adept) provides a useful technique for understanding and minimizing data traffic. Buffers between 64 and 256 words capture most environments, depending on how assignment of objects is made.

1.2 A Microprocessor Implementation of a DCA

A simple model implementing a direct correspondence architecture was simulated for a set of benchmarks. The performance was measured to guide architecture design decisions and to make a comparison with existing microprocessors. Replacing high level instructions with multiple lower level instructions caused a decrease in performance of less than 10%. This result caused the removal of instructions such as *array reference*, *for loop*, and *end for loop* instructions from the architecture. Several other statistics on instruction format frequencies and use of data types were generated. The measured performance was compared with existing microprocessors like MIPS and the M68000. Comparison is difficult, however, because of different memory and compiler/optimizer assumptions.

Memory reference caching and buffering models were evaluated for their applicability in VLSI design. By gathering data from a set of benchmarks, conclusions are being drawn for the size of on-chip data memory. Data for several on-chip data memory models are being compared together with the area occupied on the chip. The results will give insight into the usability of models which are not yet considered practical and will lead to improved combinations of different models. The partial results available show that a modified contour buffer model needs only 64 words of on-chip memory to decrease data memory bandwidth significantly. Further results will be available in the middle of 1985.

1.3 The Instruction Bandwidth of Direct Correspondence Architectures

This research involves simulation of entire families of processor architectures and simulation of cache performance for various kinds of caches. Work completed to date includes construction of a simulation workbench and initial measurement of stack and register oriented families. In preliminary work the effects of different aspects of DCAs on instruction bandwidth were measured by simulating 160 variants of the Adept architecture.

DCAs exhibit very low instruction bandwidths when compared to other classes of architectures. The contributions of various factors to this low instruction bandwidth were evaluated. The simulations were performed by modifying an Adept interpreter to estimate the instruction word fetch counts for each variation. The results indicated that the methods used by DCAs for very dense encoding of explicit operands allow DCA instructions to carry more information than instructions in other architectures without increasing the average instruction size. This accounts for most of the savings in instruction bandwidth. The methods used by DCAs to reduce the number of explicit operands (i.e. extended format sets) greatly lower instruction bandwidth requirements only when used with less dense operand encoding techniques.

1.4 Memory Hierarchies for Directly Executed Language Microprocessors

Trends in semiconductor processing technology indicate that DCA architectures are more appropriate than traditional architectures for single-chip microprocessor designs. More specifically, DCA architectures better exploit both the large amounts of memory integrated in microprocessors and the limited bandwidth available for microprocessors to access external memory. This research reports the results of studies concerning data storage requirements for DCAs and cache performance tradeoffs, thus contributing to memory hierarchy design for both DCA and other microprocessors.

The static and dynamic data storage requirements for DCA processors were examined by analyzing a set of Pascal test programs and tracing data references during execution. The operation of a buffer with a simpler structure than cache memories was specified and simulated. The results of these simulations show that a 256-word buffer typically faults on fewer than 5% of the storage allocations associated with entering procedures and captures 80% of a DCA processor's data references. Results concerning both the frequencies of different types of data references and the simulations of various buffer strategies are also reported, providing information previously unavailable to guide design tradeoffs for such microprocessors. Basically, the same 5% fault rate could be achieved with a 128 word buffer if constants were encoded in the intruction stream. Further, the 5% rate could be maintained for a 64 word buffer if, in addition, global variables were separately identified and stored.

The research also describes a model for circuit area tradeoffs in microprocessor cache designs, which differs from previous analyses in considering the overhead cost of storing address tags and replacement information along with data. Using the model, larger block sizes were found leading to better cache designs than predicted by previous studies. When the overhead cost is high, caches that fetch only partial blocks on a miss perform better than similar caches that fetch entire blocks. These results indicate that lessons for mainframe and minicomputer design practice should be critically examined for microprocessor designs.

2 Architectural Analysis

Computer architectures were compared by measuring the execution of an identical set of high level language programs. Comparative studies are difficult and expensive as they require an environment in which all the architectures can be analyzed on a common basis. Simulation has been used, but the slow speed makes it prohibitively long to collect a significant sample. Performance measures, such as the number of instructions, reflect not only architectural differences but factors (such as compilers) not related to the architecture.

The instruction streams of the IBM S/370, DEC PDP-11, and P-code machines were measured using a microprogrammable processor ("Emmy"). The measurement mechanism is embedded into the interpreter (an emulator) for the machine, and has access to all aspects of the instruction execution. The DEC VAX instruction stream was measured on a VAX 11/780 using a trace feature in the architecture. A set of FORTRAN programs was used for measurements, and reflects a scientific work load.

The analysis first studied the composition of the instruction stream. The total number of instructions executed shows the VAX architecture to be most efficient, but measures of the activity necessary by the interpreter indicate that the S/370 representation is fastest to interpret. Memory reference behavior indicated that the 8-bit displacement used by the VAX is very effective for local referencing, but VAX suffers in referencing global objects.

This work analyzed the interaction between compiler optimization techniques and the instruction streams that result from optimization. Six S/370 compilers generated different representations of the test work load, and produced the data base for study of high level language behavior and architectural analysis. Optimization, while reducing the resource demands of a program, does not apply uniformly to all aspects of instruction execution. The fixed-point computation and memory reference demands are greatly reduced, but the control requirements of a program are largely unaffected. Because the absolute occurrence of control-related instructions is constant, their relative frequency increases with optimization.

3 Concurrent Execution

The execution time of instruction can be effectively reduced by overlapping the start of the execution of one instruction with the end of the execution of another. This process of overlapping instruction execution is called *pipelining* and it is used on all "super computers". An example of an instruction stream which has been pipelined is shown in Figure 1.

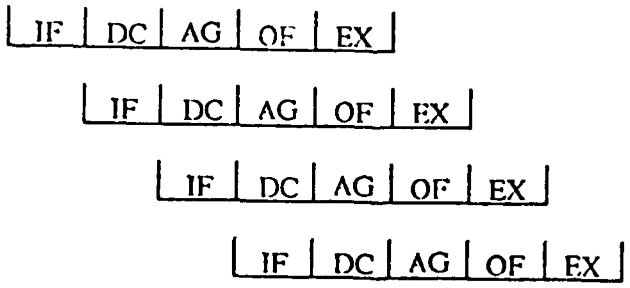


Figure 3-1: Pipelined Execution

Extending the concept of pipelining to its ultimate limit would generate an execution sequence where all instructions execute at the same time. But since simultaneous execution of all instructions would mean that inputs are fetched at the same time and results are produced at the same time, simultaneous execution could be performed correctly only if *no* instruction in the task required the completion of *any* other instruction to perform its function. Since this is possible only in the most trivial of cases, the correct execution of all other programs can occur only when instructions wait for other instructions to execute, producing results which are needed for their correct execution. An example of such an ultimate pipeline is illustrated in Figure 2. Executing multiple instructions at a time will be called *concurrent* execution of instructions as opposed to parallel execution which usually refers to a single instruction stream, multiple data stream style of computation.

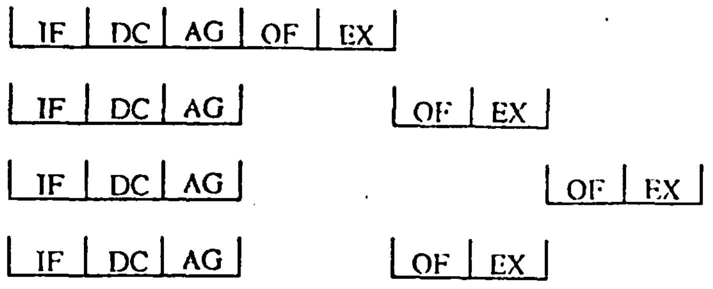


Figure 3-2: The Ultimate Pipeline

There are different degrees to which concurrent execution of instruction streams can take place. Using a minimal amount of hardware, a small amount of concurrency can be detected providing a modest increase in execution speed. The amount of concurrency detected in these schemes can be compared to the serial speedup of traditional machines which have instruction prefetch but little or no pipelining. Introducing more hardware increases the amount of concurrency which can be detected and subsequently the execution speed of the task. Concurrency such as this can be compared to a highly pipelined machine which allows out of order execution, multiple path exploration, and interleaved memory traffic.

Although the degrees of concurrency detection can actually be thought of as a continuum, four distinct levels have been defined in our research. These levels are defined so that increasing level numbers increase the amount of concurrency detected, with a corresponding increase in the amount of difficulty of detection and subsequently the amount of hardware needed for implementation. They include:

Level 0—Pipelined Execution

The main feature of Level 0 concurrency detection is that there is no concurrency actually detected at all. Level 0 concurrency is characterized by the fastest program execution possible with the single restriction that an average maximum of one instruction is executed in each machine cycle.

Level 1—Transparent Concurrency

Level 1 concurrency is characterized by a direct examination and exploitation of the concurrency which existed in the original task. In this level of concurrency, only the concurrency which was explicitly apparent in the task is detected.

Level 2—Machine Detectable Concurrency

Level 2 concurrency is marked by taking advantage of all the concurrency which a machine can detect without resorting to algorithm recoding or code manipulation.

Level 3—Algorithm Detectable Concurrency

Level 3 concurrency is characterized by analyzing the job to be done and restructuring it in hardware and software to produce a representation which will execute in a minimal amount of time using the minimum number of steps. More precisely, level 3 concurrency is all concurrency which can be detected algorithmically. This detection process can occur at both the hardware and software level.

We have completed a fairly extensive study of the speedup potential at each of the above levels. One such study[1] found the concurrency available in a sample DCA program, shown in Figure 3. Note: level 3a is compile time detection only, while level 3b represents both compile and runtime detection.

	Level of Concurrency				
	0	1	2	3a	3b
Dynamic Number of Instructions	435	435	435	390	390
Machine Cycles to Execute Task	435	306	180	121	93
Speedup	1.00	1.42	2.42	3.22	4.19

Figure 3-3: Ratios for Different Levels of Concurrency

[1]"Performance Evaluation of the Execution Aspects of Computer Architectures", by M. Flynn, J. Huck, S. Wakefield and R. Wedig, International Workshop on High-Level Language Computer Architecture, Ft. Lauderdale, FL, December 1982.

4 Participating Scientific Personnel

Professor Michael J. Flynn
Principal Investigator
Department of Electrical Engineering
Stanford University

Donald Alpert Research Assistant Stanford University	(Received Ph.D. degree in Electrical Engineering, June 1984)
Jerome Huck Research Assistant Stanford University	(Received Ph.D. degree in Electrical Engineering, March 1983)
Chad Mitchell Research Assistant Stanford University	(Ph.D. degree in Computer Science expected November 1985)
Johannes Mulder Research Assistant Stanford University	(Ph.D. candidate, Electrical Engineering)
Evan Tick Research Assistant Stanford University	(Ph.D. candidate, Electrical Engineering)
Scott Wakefield Research Assistant Stanford University	(Received Ph.D. degree in Electrical Engineering, December 1982)
Robert Wedig Research Assistant Stanford University	(Received Ph.D. degree in Electrical Engineering, June 1982)
Andrew Zimmerman Research Assistant Stanford University	(Ph.D. candidate, Electrical Engineering)

5 Technical Reports and Publications Sponsored Under Contract

DEL Formats for Recursive Languages
by J. D. Johnson
CSL Technical Note 202, May 1982

"Detection of Concurrency in Directly Executed Language Instruction Streams"
by Robert G. Wedig, Ph.D. Thesis, Stanford University, June 1982.
(Also available as CSL Technical Report No. 238, January 1983)

"Concurrency Detection in Language-Oriented Processing Systems"
by M. Flynn and R. Wedig
Proceedings of the 3rd International Conference on Distributed Computing Systems
Miami/Ft. Lauderdale, FL, October 1982

"Performance Evaluation of the Execution Aspects of Computer Architectures"
by M. Flynn, J. Huck, S. Wakefield, and R. Wedig
Proceedings of the International Workshop on High Level Language Computer Architecture, December 1982, Ft. Lauderdale, FL.

"Studies in Execution Architectures"
by Scott Wakefield, Ph.D. thesis, Stanford University, December 1982.
(also available as CSL Technical Report No. 237, January 1983)

"A Local Variable Storage Mechanism"
by Scott Wakefield
COMPCON Proceedings March 1983, San Francisco, CA.

"Execution Architecture: The DELtran Experiment"
by M. Flynn and L. Hoevel
IEEE Transactions on Computers, C-32(2):156-175, February 1983.

"Comparative Analysis of Computer Architectures" p
by Jerome Huck, Ph.D. thesis, Stanford University, March 1982.
(also available as CSL Technical Report 83-243, May 1983)

"Comparative Analysis of Computer Architectures"
by J. Huck and M. Flynn
Proceedings of the 9th World Computer Congress, IFIP
Paris, September 1983.

"Stanford Emulation Laboratory"
by M. Flynn
SIGmicro Newsletter, 14(3):10-17, September, 1983

"Directly Executed Language Architectures for VLSI Processor Design"
by D. Alpert, M. Flynn, and S. Wakefield
Proceedings, International Conference on Computer Design, pp. 609-612
October, 1983.

Data Buffers for Execution Architectures
by D. Alpert
CSL Technical Report 83-250, November 1983

"Towards Better Instruction Sets"
by M. Flynn
SIGmicro Newsletter, 14(4):3-8, December, 1983

[2]Evaluation of an Interpreted Architecture for Pascal on a Personal Computer
by C. Mitchell
CSL Technical Report 83-253, December 1983

Performance Tradeoffs for Microprocessor Cache Memories
by D. Alpert
CSL Technical Note 83-239, December 1983

Memory Hierarchies for Directly Executed Language Microprocessors
by D. Alpert, Ph.D. thesis, Stanford University, June 1984
(also available as CSL Technical Report 84-260, June 1984)

"Measures of Ideal Execution Architectures"
by M. Flynn and L. Hoevel
IBM Journal of Research and Development 28(4):356-369, July 1984

[2] Instruction Bandwidth of Direct Correspondence Architectures
by C. Mitchell
CSL Technical Report 84-267, December 1984

"On Instruction Sets and Their Formats"
by M. Flynn, J. Johnson, and S. Wakefield
IEEE Transactions on Computers, C-34(3):242-254, February 1985

"Adept—A Pascal Based Architecture"
by S. Wakefield and M. Flynn
(revised version submitted to ACM Transactions on Computer Systems, January 1985.)

[2]These reports were produced under an IBM Fellowship, based on work continuing from ARO sponsored projects.